Implementing set-theoretic relational-query functions using highly parallel index-processing hardware

by SAKTI PRAMANIK
Michigan State University
East Lansing, Michigan

ABSTRACT

Hardware organizations for processing set-theoretic database query functions are presented. These organizations implement the functions by processing index trees. One advantage of this approach is that the index trees can be merged in a highly parallel fashion. Hardware organizations proposed here use the database machine approach, thus processing the index-trees on the fly. Experimental results giving the performances of these organizations are presented. Finally, a slight variation of the index tree representation, requiring much less storage for the index, is given.
INTRODUCTION

One important advantage of relational data models is that a relation is treated as a set of tuples, so a set-theoretic database query language can be designed for it. These set-theoretic database query functions are a powerful tool for the database's users. But the implementation of such functions on a von Neumann architecture is not efficient. This inefficiency is caused by the disk input/output (I/O) bottleneck and the lack of multiprocessing capability. Several database machine architectures and architectures for very-large-scale integration (VLSI) have been proposed. They all have multiprocessing capability. But the problem of disk I/O bottleneck for a large database still remains. This paper proposes some hardware organization suitable for implementing set-theoretic query functions. The architecture proposed here is based on processing tree type index structure. This processing is done on the fly as the indexes are being read off the secondary storage.

STRUCTURE OF INDEX TREES AND THEIR USE IN IMPLEMENTING SET-THEORETIC QUERY FUNCTIONS

One of the many ways of representing an index for a file (a set of entities) is by storing the keys in a tree structure. An example of such an index tree is given in Figure 1. Here every character of the key is represented by a node of the tree. There are two distinct advantages of structuring an index in this fashion. First, this is a more compressed representation of an index than storing each key separately would produce. Second, and more important, this tree structure allows a convenient and fast way of merging two or more indexes; thus it provides an efficient way of implementing set-theoretic database query functions, such as intersection, difference, and union. For example, the intersection function can be implemented by traveling through the index trees of both relations, and comparing their nodes. This is shown in Figure 2. Here we compare the root nodes of both trees, and find that only the character A matches. In the next level, only the children of node A need to be compared. The interesting fact is that we can discard all the subtrees under the nodes that do not match. Further, the algorithm to merge two or more index trees, can be highly parallel. The merger is sequential along the depth of the tree and parallel along its width.

DIFFICULTIES OF MERGING INDEX-TREES IN PARALLEL

One difficulty of merging these index-trees is that a tree itself can be very big, big enough that only a part of it can be fetched into main memory at a time. Another is that it is difficult to exploit the inherent parallelism mentioned above, because the many subtrees that are to be merged in parallel need to be determined first, dynamically, and then loaded into the processors in parallel. We will give a database machine approach to solve this problem, where the subtrees will be determined first and then merged on the fly, as they are being read off the disk. Before we discuss this approach in detail, we would like to mention that a hierarchical sequential-storage structure will be used for storing the subtrees of an index. An example of hierarchical sequential storage is given in Figure 3. Here a tree is represented by a linear list of nodes of the index.
tree. Thus we can merge two index-trees by scanning the corresponding two lists, sequentially. The advantage of this storage structure is that the subtrees are readily available, right next to their parent node, in the same memory block. Also, the twin pointers help in skipping portions of the list, which accelerates the merger process.

SEQUENTIAL MERGING

The number of nodes of an index-tree of the names of 50,000 randomly chosen persons has been computed; it was found to be about 1,000,000. Though this is only an estimate, unloading data of this magnitude requires a lot of disk accesses. We will process data directly on the disk, as they are being read from it. First, we will consider sequential processing, requiring only one data stream per relation. Thus, to merge two relations we will have two data streams, each representing the nodes of the index-tree in a hierarchical sequential fashion. Each node contains the character and its level number in the tree. The level numbers are coded by a single bit as follows. A 0 indicates that the level number is one higher than the preceding node’s. A bit 1 indicates that the number following is the level number of the node. An illustration of this coding scheme is given in Figure 4. The advantage of this coding scheme is that the nodes are stored in a hierarchical sequential order; thus the level numbers increase consecutively in the order in which the nodes are stored.

We use the algorithm of Figure 5 to merge two such streams. The twin pointers mentioned before are not needed, because the scanning will be sequential in a stream. In the figure, level 1 and level 2 represent the levels of the current nodes being compared in stream 1 and stream 2, and SCAN 1 and SCAN 2 represent the functions to get the next node of stream 1 and stream 2. Ch1 and Ch2 are the characters being compared.

One problem of merging the data streams at the rate of data flow is that we need to freeze a node of one stream when we compare it against a corresponding node in the other stream that has not arrived yet. Since both trees are ordered, all the intervening nodes in the second stream are discarded. We can freeze a node by using some buffer memory. The size of this buffer memory, in the worst case, must be as big as the largest index-tree; though the probability of this worst case happening is very low. We can reduce this buffer to any arbitrary size by stopping the data flow of an incoming stream when the buffer overflows. The data flow is allowed to start again when the buffer becomes available. For a disk type storage device this means a loss of a few revolutions. There is a tradeoff between the buffer size and the number of revolutions lost. We could, however, reduce this time loss by clustering the children nodes as shown in Figure 6. The advantage of this clustering is that the clustered nodes are the twin nodes, and all of them are available for comparison without any loss of revolution. Of course, now we will need to freeze a cluster but this will require less buffering because some tracks may now be skipped altogether.

PARALLEL MERGING

Instead of merging the index-trees in a hierarchical sequential fashion, we could also merge them level by level. Thus, we compare the characters of level 1 of both index-trees, first. Next we compare the children of these matched nodes, and so on. One serious problem of implementing this algorithm is to select from storage only the children of these matched nodes, and load them into the appropriate processors. Parallel loading is important, because the amount of data to be loaded can be very large.

Here again we will use a database machine approach to merge the index-trees. We will do this by storing an index-tree
in secondary memory, in two parts, and processing each part separately. The first part consists of the first K levels of the index-tree, starting with the root level. The second part contains the rest of the nodes. This is shown in Figure 7. Thus we will have as many index-subtrees in the second part as there are leaf nodes in the first part. We will store these index subtrees on the disk tracks for processing in parallel, on the fly. The extent of this parallel operation depends on the number of index subtrees there are in the second part. On the other hand, if we can read N tracks from the storage, simultaneously, there can be N parallel data flows, giving us a maximum of N parallel processing.

In merging two index trees we will first compare the first part of both the indexes. The size of the first part will be rather small and can be merged by fetching the page containing it. Typically the first part will consist of the first three or four levels, containing only about 500 to 2,000 nodes. (For a 26-character alphabet, the maximum number of nodes ever possible will be 26 for level 1 and 676 for level 2.) By merging these nodes of the first part we are able to determine which index subtrees in the second part are to be merged. We then load the appropriate instructions into the track processors for merging these index-subtrees. We assume a processor per track architecture for merging the second part. Each index subtree will be processed sequentially; they are stored on the tracks in hierarchical sequential fashion.

There are two problems with this processor-per-track architecture. The first problem is that the two index subtrees to be merged may reside on two different tracks. We need data-paths to transfer index subtrees of one track into the processor of another. Second, the two subtrees to be merged may not arrive at a processor at the same time. We solve this problem by using some buffer memory in each processor; it will be shown that the buffer size is reduced considerably by ordering the index-subtrees on the tracks, and by a proper track allocation strategy, that is, how subtrees are stored across the tracks.

**DATAPATH BETWEEN ADJACENT PROCESSORS AND DISTRIBUTED TRACK ALLOCATION**

Two different datapath organizations, each suitable for a particular type of track allocation scheme, are presented. The first organization provides a datapath between each adjacent processor. This is shown in Figure 8. When an index subtree is read from a track, it is transferred to the appropriate processor through this path. To avoid any path overlap of two simultaneous transfers, we schedule the merging of index subtrees. For example, there is an overlap of path 1 and path 2 in Figure 8, because the path through processor i is shared by both. We avoid this overlap by scheduling the merger, using path 1 first and then path 2 in the second revolution. We are able to schedule them because the track position of each of the index subtrees is known. This serialization process increases the number of disk revolutions required. But this again can be minimized considerably by ordering the index subtrees on the tracks, as well as by using a distributed track allocation scheme. In distributed allocation, all the subtrees of an index-tree are spread across as many tracks as possible. Ideally, in distributed allocation, the first K subtrees will be on the first track, the next K subtrees on the second track, and so on, where K is the ratio of the number of subtrees to the number of tracks available. Table I shows the dependence of the selectivity ratio* and the value of K on the data path congestion. It is seen that the data path congestion increases with selectivity ratio initially, but starts decreasing again with further increase in selectivity ratio. The reason for this decrease is that the higher selectivity ratio means more likelihood of the matching subtrees being resident on adjacent tracks or on the same track. In this experiment we avoided data path congestion by scheduling only one subtree per track per revolution and having no two tracks with overlapping paths in the

![Figure 7](image7.png)  
**Figure 7**—Partitioning index tree into two parts and creating subtrees for parallel processing

![Figure 8](image8.png)  
**Figure 8**—Hardware organization using data path between adjacent processors

<table>
<thead>
<tr>
<th>Selectivity Ratio</th>
<th>K = 1*</th>
<th>K = 2</th>
<th>k = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>.025</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>.125</td>
<td>7</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>.250</td>
<td>10</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>.500</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>.750</td>
<td>10</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>.880</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

*For K < 1, the number of disk revolutions could be further reduced.

*The selectivity ratio is the ratio of the number of joined tuples to the number of tuples in the joining relations.
same revolution. We could, however, schedule more than one subtree on a track in the same revolution if we knew the relative rotational positions of the subtrees. This would have reduced the number of revolutions required even further.

The worst-case buffer requirement on each track depends on the number of subtrees on a track that belong to the same index-tree. By distributing them over many tracks, we are able to reduce the size of the worst-case buffer requirement. Further, this distribution also helps in reducing the number of subtrees competing for a particular segment of the path. This helps in reducing the data path congestion also.

TRANSFER PATH BY COMMON BUFFER POOL AND CLUSTERED TRACK ALLOCATION

Though this distributed track allocation helps in reducing the worst-case buffer size as well as the data path congestion, distributing the subtrees uniformly over all the tracks may be a problem. This uniform distribution is important because it reduces the amount of buffer required per track. This is discussed in the next section. Second, path congestion may still be a problem for large numbers of subtrees.

On the other hand, transfer path by common buffer pool helps in reducing this congestion problem by clustering the subtrees within a few tracks. In this allocation scheme we store the subtrees of an index tree on the same track, and if it overflows, we allocate another track, and so on. Thus we will have as few tracks as possible for storing the subtrees of a single index tree. We use a different mechanism for data transfer between tracks here, because at any one revolution only these few tracks will be passing data back and forth, as opposed to all the tracks in the distributed track allocation.

The hardware organization for data transfer path is given in Figure 9. Here we use a common transfer path between all the processors and schedule this common path sequentially among the processors by a circular priority line. Each processor keeps control of the path for no more than a preset maximum time period, and passes control to the next processor as soon as it is done with its transfer. When a processor gets control of the path, it can read from or write into a random-access buffer memory through this path. The idea here is to collect the subtrees and save them in the buffer pool until the corresponding subtrees have arrived in a processor, at which point we can start merging them in this processor. One advantage of this scheme is that we can use a common buffer pool instead of having separate buffer pools, statically, before the actual processing on the disk starts. We load each processor with the addresses in the common pool where these subtrees will be stored. Of course, these addresses will be overlapping because the memory for the subtrees that have been completely processed in the early part of the disk revolution can be reused in the later part.

A TRADEOFF BETWEEN THE SPEED OF THIS TRANSFER PATH AND THE NUMBER OF REVOLUTIONS REQUIRED

One serious problem of using such a path is the high transfer rate it requires. For example, if there are \( N \) tracks containing the subtrees of the two index trees to be merged, we will in the worst case need a path that is \( N \) times faster than the transfer rate of each of these tracks. Though the probability of this situation occurring is rather low, we can use a relatively slow transfer path by processing only a few tracks at a time. This will need a few extra revolutions to complete the merger of all the subtrees. Another advantage of processing a few tracks at a time is that the size of the buffer pool required will be even smaller.

EXPERIMENTAL RESULTS TO COMPARE THE PERFORMANCES OF THESE TWO APPROACHES OF DATA TRANSFERS

Experiments were performed to measure and compare the performances of these two organizations. We simulated the two systems with the following assumptions: only one of the two subtrees to be merged needs to be transferred to another processor; each subtree is assumed to be of the same size. These are rather simplifying assumptions, and the actual size of these subtrees depends on such factors as the type of database and the number of levels in the subtrees. We also assume that the pointers to the tuples are stored following each key in the index tree.

Table II shows the amount of common buffer pool needed

<table>
<thead>
<tr>
<th>Selectivity Ratio</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( K=1 )</td>
<td>( K=2 )</td>
<td>( K=4 )</td>
<td></td>
</tr>
<tr>
<td>.025</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>.125</td>
<td>7</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>.250</td>
<td>10</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>.500</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>.750</td>
<td>10</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>.880</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Note: The value of each entry in the table represents the number of disk revolutions. For \( K = 1 \), this value could be further reduced. (The number of tracks was assumed to be 480 in the experiment.)

Figure 9—Data transfer by a transfer bus
for different selectivity ratios. First of all, it is interesting to see from column 2 that only 9.5% of the subtrees storage of an index is needed for the buffer pool. This size is very small compared to the total index database size. It is also interesting to note that the buffer pool size increases with decreasing number of subtrees per track. This is because the number of tracks required to store the subtrees increases but the buffer required per track does not decrease in proportion. We also see from this table that the buffer-pool size increases with increasing selectivity ratio at first, and then starts decreasing again with a further increase in selectivity ratio. The reason for this decrease is that the higher selectivity ratio means more likelihood of two matching subtrees being in the same relative positions on the tracks.

In distributed track allocation the buffer size per track is equal to the largest possible storage needed on a track for all subtrees of a single index. Thus, if we can distribute the subtrees of an index uniformly over the tracks, it will result in a reasonably sized buffer. Wide variations in subtree sizes, however, make this uniform distribution difficult. We can force these subtree sizes to be more uniformly distributed by making the subtrees start at a higher level in the index tree. This will create more subtrees, each one being smaller in size.

The distributed track approach will require more buffer than the clustered approach because the former allocates buffer statically among the tracks. The storage requirement as well as the data path congestion can be reduced considerably if extra revolutions are allowed. This was shown in Figure 9. Extra revolutions also help the clustered approach in reducing the buffer size and the memory speed requirement.

**BINARY INDEX TREE**

A slight variation of the index tree approach, mentioned earlier, will now be described. This approach requires significantly less storage, and the underlying data structure is still a tree. Further, it can be a basis for parallel merging along the width, as well as along the depth of the tree. This new tree will be called the binary index tree. We will first describe the mechanics of this binary-index-tree, and then show by experimental results the significance of this approach in saving a large amount of storage. Finally, in the Appendix, we will present a proof of this result.

It has been seen that an index tree fans out more in the first few levels of the tree than in later levels. Consequently, the index tree will have a large width at higher levels, which will therefore have a lot of nodes. We can reduce the number of nodes by partitioning the index tree vertically, as shown in Figure 10, and forming independent trees for each partition. The tree size for each partition will be very small, but we will now need a mapping between these trees. We accomplish this mapping by adding extra nodes to these trees at a logarithmic rate. These added nodes contain pointers to the corresponding nodes in the other trees. For example, an index tree can be split vertically, as shown in Figure 10, with each partition having two levels. Thus, an index tree with $M$ levels will produce $M/2$ smaller trees. Each one of these trees, in fact, will be an index tree for keys, as if the keys are each of length two; the first two characters of each key produce tree one, the next two characters produce tree two, and so on. Now we map a pair of these trees at a time by adding an extra level of nodes to one of them. Each of these nodes points to the leaf nodes of the other tree in the pair. Thus, we have $M/4$ trees with three levels. Next, we map these $M/4$ trees only, again a pair at a time, creating $M/8$ trees, with four levels each. We continue this process until we have one tree with $(\log_2 M) + 1$ levels. An example of this is shown in Figure 10.

**A SIMPLIFIED MODEL AND EXPERIMENTAL RESULTS**

We assume that the average fan-out of a node at the $i$th level is $N_i + 1$. The number of nodes at the root level (i.e., level 1) is $N_1$, say. Then in the $i$th level the number of nodes will be $N_1 \times N_2 \times \ldots \times N_i$. Thus the total number of nodes, $S$, for an index tree $M$ levels deep, is:

$$S = \sum_{i=1}^{M} \left( \prod_{j=1}^{i} N_j \right).$$

The number of nodes, $S_1$, for a binary index tree will be

$$S_1 = N_1 M + \sum_{i=1}^{M} \frac{M}{2^i} \left( \prod_{j=1}^{i} N_j \right).$$

Using these two formulas we have computed the values of $S$ and $S_1$, for different values of the sequences $N_1 \times N_2 \times \ldots \times N_M$. These are given in Table IV. Each group size is defined in Table III. It is seen that the amount of storage needed in binary index tree is less than in the index tree approach. Further, this saving increases with increasing length of the keys. For example, the saving is 66.4% for keys of length 16, and is 72.53% for keys of length 32. It is also seen from Table IV that the percentage gain for the binary index tree is less for faster growth of the sequences $N_1, N_2, \ldots, N_M$.

It is important to note here that the number of nodes required in the binary index tree approach is considerably less but some of the nodes need extra storage for the mapping. It turns out, though, that the amount of this extra storage is much less than the large amount saved by reducing the number of nodes. Finally, it is shown in the appendix that for any value of the sequence $N_1, N_2, \ldots, N_M$, the number of nodes for the binary index tree is less than or equal to the number of nodes for the index-tree.
TABLE III—Average fanout at each level of the index tree of a group

<table>
<thead>
<tr>
<th>Group#</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>N4</th>
<th>N5</th>
<th>N6</th>
<th>N7</th>
<th>N8</th>
<th>N9</th>
<th>N10</th>
<th>N12</th>
<th>N13</th>
<th>N14</th>
<th>N15</th>
<th>N16–N64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26</td>
<td>11</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>1.4</td>
<td>1.2</td>
<td>1.1</td>
<td>1.01</td>
<td>1.1</td>
<td>1.05</td>
<td>1.1</td>
<td>1.05</td>
<td>1.2</td>
<td>1.045</td>
</tr>
<tr>
<td>2</td>
<td>26</td>
<td>11</td>
<td>8</td>
<td>4</td>
<td>1.44</td>
<td>1.25</td>
<td>1.3</td>
<td>1.1</td>
<td>1.01</td>
<td>1.1</td>
<td>1.05</td>
<td>1.1</td>
<td>1.15</td>
<td>1.05</td>
<td>1.02</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td>12</td>
<td>8</td>
<td>4</td>
<td>2.4</td>
<td>1.4</td>
<td>1.1</td>
<td>1.1</td>
<td>1.05</td>
<td>1.1</td>
<td>1.05</td>
<td>1.1</td>
<td>1.1</td>
<td>1.113</td>
<td>1.1</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>18</td>
<td>10</td>
<td>5</td>
<td>1.5</td>
<td>1.1</td>
<td>1.2</td>
<td>1.08</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>5</td>
<td>26</td>
<td>18</td>
<td>12</td>
<td>8</td>
<td>1.13</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
</tbody>
</table>

*An index with 50,000 keys is assumed.

APPENDIX

Proposition The number of nodes, $S$, of an index tree is greater than or equal to the number of nodes, $S_1$, of a binary index tree for any values of the sequence $N_1, N_2, \ldots, N_M$.

Proof. From (1) and (2) we have

$$S = \sum_{i=1}^{M} \prod_{j=1}^{i} N_j$$

$$S_1 = N_1 \frac{M}{2} + \sum_{i=1}^{\log_2 M} \frac{M}{2} \prod_{j=1}^{2i} (N_j).$$

In general, for a large number of keys, $S$ and $S_1$ are fairly large, and we ignore the first term from both $S$ and $S_1$. Thus we have

$$S = \sum_{i=2}^{M} \left( \prod_{j=1}^{i} (N_j) \right)$$

$$S_1 = \sum_{i=1}^{\log_2 M} \left( \frac{M}{2} \prod_{j=1}^{2i} (N_j) \right)$$

Note that the last terms of $S$ and $S_1$ are the same. Since we want to compare the values of $S$ and $S_1$, we can ignore this similar term from both. Thus from $S$ we get

$$S' = \sum_{i=2}^{M-1} \left( \prod_{j=1}^{i} (N_j) \right)$$

and from $S_1$ we get

$$S_1' = \sum_{i=1}^{\log_2 M-1} \left( \frac{M}{2} \prod_{j=1}^{2i} (N_j) \right).$$

Now $S'$ can be rewritten as

$$S' = \sum_{i=1}^{\log_2 M-1} \left( \prod_{j=1}^{2i} (N_j) \right) \left( 1 + \sum_{i=1}^{\log_2 M-1} \sum_{k=2i+1}^{2i} (N_k) \right)$$

Thus the term $\prod_{j=1}^{i} N_j$ is same as in $S_1'$. We abbreviate this term as $C_i$. Thus,

$$C_i = \prod_{j=1}^{i} N_j.$$

It should be noted that $C_i \leq C_{i+1}$ for all $i$, because $N_i \geq 1$ for all $i$.

Now we can write $S'$ and $S_1'$ as

$$S' = \sum_{i=1}^{\log_2 M-1} C_i \left( 1 + \sum_{i=1}^{\log_2 M-1} \sum_{k=2i+1}^{2i} (N_k) \right)$$

$$S_1' = \sum_{i=1}^{\log_2 M-1} \frac{M}{2} C_i.$$

The term within parentheses for $S'$ satisfies the following condition,

$$1 + \sum_{i=1}^{\log_2 M-1} \sum_{k=2i+1}^{2i} (N_k) \geq 2^i,$$

because there are $2^i - 1$ terms in the summation, each of which is greater than or equal to one. Thus we have

$$S' \geq S'' = \sum_{i=1}^{\log_2 M-1} 2^i C_i \quad \text{(A1)}$$

By subtracting $S_1'$ from $S''$ we get

$$S'' - S_1' = \sum_{i=1}^{\log_2 M-1} C_i \left( 2^i - \frac{M}{2} \right)$$

The absolute values of $2^i - M/2$ for $i = 1$ through $(\log_2 M) - 1$ form a duplicating sequence. For example, its values for $i = 1$ and $i = (\log_2 M) - 1$ are $2 - M/2$ and $-(2 - M/2)$, respectively. Similarly, for $i = 2$ and $i = (\log_2 M) - 2$ the values are $(4 - M/4)$ and $-(4 - M/4)$, respectively, and so on. Thus, we can write

$$S'' - S_1' = \sum_{i=1}^{N} \left( \frac{M}{2} - 2^i \right) \left( C_{2N+1-i} - C_i \right)$$

where $N = (\log_2 M - 1)/2$.

For this range of values of $i$ we have

$$(M/2 - 2^i) \geq 0$$

and

$$(C_{2N+1-i} - C_i) \geq 0.$$
Thus

\[ S'' - S'1 \geq 0. \]  \hspace{1cm} (A2)

From relations (A1) and (A2) we get

\[ S' - S'1 \geq 0, \]

which was to be demonstrated.

REFERENCES
